An Improvement in Partial Order Reduction Using Behavioral Analysis

Yingying Zhang, Emmanuel Rodriguez, Hao Zheng, and Chris Myers

***Abstract*— Efficacy of partial order reduction in reducing state space relies on adequate extraction of the independence relation among possible behaviors. However, traditional approaches by statically analyzing system model structures are often not able to reveal enough independence for reduction. To address such a problem, this paper presents a behavioral analysis approach that uses a compositional reachability analysis method to generate the over-approximate local state spaces for all modules in a system where a much more precise independence relation can be extracted for partial order reduction. Compared to the static analysis approaches, significantly higher reduction on complexity can be seen in a number of non-trivial examples, and as a consequence, dramatically less time and memory are required to finish these examples.**

***Index Terms*— formal verification, model checking, partial order reduction, state space analysis, compositional reasoning.**

I. Introduction

Model checking is a formal automated analysis method for verifying hardware and software systems. It systematically checks whether a model of a given system satisfies a desired property such as deadlock freedom and request-response properties [2]. Model checking has been developed into a mature and widely used approach in many applications. For an extensive review of model checking, please refer to [2], [5].

For synchronous systems, all state variables are updated simultaneously due to the global control of a clock. Instead of sharing a common clock and exchanging data on clock edges, asynchronous designs communicate through control protocols, and multiple components can perform executions concurrently. When verifying such a system, concurrently enabled executions need to be interleaved so that all possible orderings of executions are considered to avoid missing any behavior. The need to consider all possible interleavings of concurrent executions is the main cause of state explosion as the number of interleavings grows exponentially if a system has a high degree of concurrency, and this leads to an excessively large state space for even a relatively small system.

In order to address state explosion, partial order reduction (POR) have been developed [16], [7], [3], [6], which focuses on reducing states that do not affect the final verification results. Instead of executing all enabled transitions in a state, POR finds an $ample$ set [5] which is a subset of the full enabled set. This can lead to an exponentially smaller state space while still representing sufficient behavior for verification to derive the same results as on the original full state space. Identifying unnecessary interleavings among the enabled transitions plays an important role in POR which is based on examining the dependency relations that exist between the transitions of a system. Since calculating the precise dependency relation between transitions may be as hard as verifying the whole system, POR often uses a conservative statically computed approximation for the dependency relation of the transitions [16], known as static POR, which guarantees as a priori to include all relevant computations of the system. Dynamic POR [3], [7], which excludes the need to apply static analysis a priori by detecting data dependencies dynamically, may potentially miss certain relevant paths which can be added at a later step. These approaches can potentially incur very high runtime overhead.

Computing independence relations among the concurrent executions is critical for the effectiveness of POR. As indicated above, existing static approaches may not be able to generate sufficiently accurate independence relations while the dynamic approaches can have very high runtime overhead. To address these problems, this paper presents a behavioral analysis method to compute an independence relation among the concurrent executions. This behavioral analysis method relies on an efficient compositional reachability analysis approach that generates the over-approximated state space models for all modules in a system [18], and extract an independence relation from the generated state space models. The independence relation found this way is more refined and accurate compared with the static analysis based approaches, which leads to more reduction than allowed by the static approaches.

After an independency relation is obtained, another key step is to produce a subset of enabled transitions, often referred to as *ample set*, from the full enabled set in every state. Ideally, the ample set generated in each state is minimized to achieve the maximal reduction. To meet this requirement, this paper also introduces two new rules for ample set generation. One rule focuses on how to choose independent transitions to construct a small ample set quickly while avoiding loss of any essential interleavings of transition executions. Another rule considers the cycle condition [5] such that the size of the ample sets can be reduced even further.

This material is based upon work supported by the National Science Foundation under Grant No. 0546492 and 0930510. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Yingying Zhang, Emmanuel Rodriguez and Hao Zheng are with the Dept. of Computer Science and Engineering at the University of South Florida, Tampa, FL. Chris Myers is with the Dept. of Electrical and Computer Engineering at the University of Utah, SLC, UT.

978-0-7695-4767-1/12 $26.00 © 2012 IEEE
DOI 10.1109/ISVLSI.2012.15

II. Background

A. Labeled Petri-Nets

This paper uses *Labeled Petri-Nets* to model asynchronous systems. Petri-Nets are a common modeling formalism for asynchronous designs [12], [1]. A Petri-net is a directed graph with a set of transitions and a set of places. A labeled Petri-Net is a Petri-net where transitions are labeled with various information representing a system's properties and behavior [14]. Its definition is given as follows.

Definition 2.1: A labeled Petri-net (LPN) is a tuple $N = \langle V, P, T, F, M_0, init, L\rangle$,where

1) V is a set of state variables of the integer type,
2) P is a finite set of places,
3) T is a finite set of transitions,
4) $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of the flow relations,
5) $M_0 \subseteq P$ is a finite set of initially marked places,
6) $init : V \to \mathbb{Z}$ is a labeling function that assigns each variable an initial value,
7) $L = \langle Guard, Assign\rangle$ is a pair of labeling functions for transitions in T, which is defined below.

A simple LPN example is shown in Fig 1. Fig.1(a) shows a simple asynchronous circuit consisting of three components, and Fig. 1(b) shows the LPNs for each component in the circuit. For each component, its LPN has 4 places and 4 transitions. The places are represented as circles, and the transitions are represented as boxes. Each place is preceded and followed by one or more transitions, and each transition is preceded and followed by one or more places. The flow relations are represented by the edges connecting the transitions and places [1]. The bullets found in some places are called tokens. Each place can have at most one token. A place is *marked* if it has a token. A marking of LPN, $M \subseteq P$ is a set of marked places.

The dynamic behavior of a concurrent system is captured by LPN transitions with labelings. Each transition $t \in T$ has a $preset$ denoted by $\bullet t = \{p \in P | (p,t) \in F\}$, which is the set of places connected to t, and a $postset$ denoted by $t\bullet = \{p \in P | (t,p) \in F\}$, which is the set of places to which t is connected. The $preset$ and $postset$ for places are defined similarly.

Before defining the transition labels formally, the grammar used by these labels is introduced first below [14]. The numerical portion of the grammar is defined as follows:

$$\begin{aligned}\chi \quad ::= \quad & c_i \mid v_i \mid (\chi) \mid -\chi \mid \chi + \chi \mid \chi - \chi \mid \chi * \chi \mid \\ & \chi / \chi \mid \chi \hat{} \chi \mid \chi \% \chi \mid \text{NOT}(\chi) \mid \text{OR}(\chi, \chi) \mid \\ & \text{AND}(\chi, \chi) \mid \text{XOR}(\chi, \chi) \mid \text{INT}(\phi)\end{aligned}$$

where c_i is an integer constant from $\mathbb{Z}$, and v_i is an integer variable. The functions NOT, OR, AND, and XOR are bit-wise logical operations, and they are only applicable to integers and assume a 2's complement format with arbitrary precision. INT(ϕ) returns 1 if the Boolean expression ϕ, which is defined below, evaluates to **true**, or 0 otherwise. The set $\mathcal{P}_\chi$ is defined to be all formulas that can be constructed from the χ grammar.

The Boolean portion of the grammar is as follows:

$$\begin{aligned}\phi \quad ::= \quad & \textbf{true} \mid \textbf{false} \mid v_i \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \chi \equiv \chi \mid \\ & \chi \geq \chi \mid \chi > \chi \mid \chi \leq \chi \mid \chi < \chi\end{aligned}$$

where the integer v_i is regarded as **true** if its value is nonzero, and **false** otherwise. In this sense, it is similar to the semantics of the C language. The set $\mathcal{P}_\phi$ is defined to be all formulas that can be constructed from the ϕ grammar.

As in Definition 2.1, each LPN transition is labeled with an enabling condition and a set of variable assignments. LPN transition labeling is defined by $L = \langle Guard, Assign\rangle$ where

- $Guard : T \to \mathcal{P}_\phi$ labels each LPN transition with a Boolean expression that defines its enabling condition.
- $Assign : T \times V \to \mathcal{P}_\chi$ labels each LPN transition $t \in T$ and each variable $v \in V$ with an integer assignments made to v when t fires.

B. Reachability Analysis

A basic approach for analyzing the dynamic behavior of a concurrent system modeled with LPNs is reachability analysis, which finds all possible state transitions and thus reachable states for such a system. The reachable state space is typically represented by a state graph. State graph (SG) is a directed graph where vertices represent states and edges represent state transitions.

The state of the LPNs is the pair (M, σ) where M denotes the marking and σ denotes the vector of variable values. Given a state s, $M(s)$ is a set of marked places in s and $\sigma(s)$ is the state vector of s. Also, for any expression $e \in \mathcal{P}_\chi \cup \mathcal{P}_\phi$, $value(e, s)$ denotes a function that returns the value of expression e in state s.

Before describing the reachability analysis approach, the LPN enabled transition is defined below:

Definition 2.2 (Enabled Transition): A LPN transition t is enabled in a state s if the following two conditions are met:

1) $\bullet t \subseteq M(s)$,
2) $value(e, s)$ is $true$ or not zero for $e = Guard(t)$.

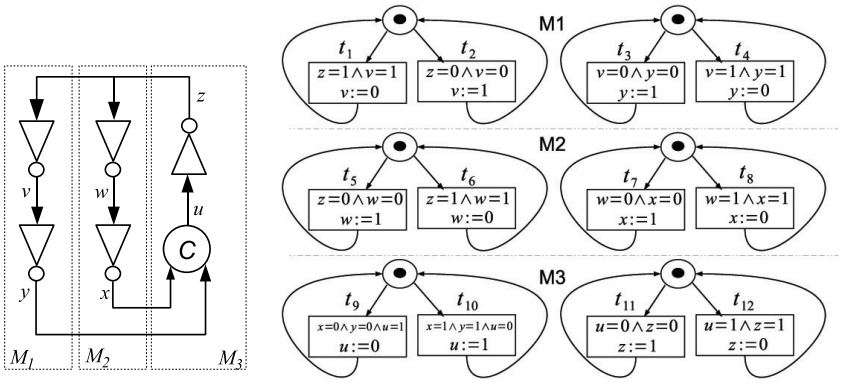
In Fig 1, every transition has its preset included in the initial marking. In the initial state, the values of variable u and z are 0, $Guard(t_{11})$, which is $u = 0 \wedge z = 0$, is evaluated to be true, therefore transition t_{11} is enabled in the initial state.

Given a LPN model, the set of transitions enabled in a state s is denoted by $enabled(s)$. The reachable state space of a LPN model can be found by exhaustively firing every enabled transitions starting at the initial state. Firing a transition leads to a new state by generating a new marking and a new state vector according to the assignments labeled for such a transition. Detailed definition of transition firing can be found in [1]. In this paper, $s' = t(s)$ denotes that a new state s' is produced by firing transition t in state s.

The procedure to find the reachable state space of a given LPN model is given in Algorithm 1.

III. Computing Independence Relations

From the previous section, the behavior of a concurrent system is modeled by executing transitions enabled in a state.



(a) (b)

Fig. 1. (a) A simple asynchronous circuit. (b) The LPNs for module M_1, M_2, and M_3. The initial values of variables u, v, w, x, y, and z are 0, 1, 1, 0, 0, and 0, respectively.

Algorithm 1: $search((T, P, F, M_0))$

Output: Reachable number of state in stateTable

```
1  stateStack.push(s_0);
2  enabledStack.push(enabled(s_0));
3  while stack is not empty do
4      s = stateStack.top();
5      E_s = enabledStack.top();
6      if E_s = ∅ then
7          stateStack.pop();
8          enabledStack.pop();
9          continue;
10     Select t ∈ E_s to fire;
11     enabledStack.push(E_s \ t);
12     s' = t(s);
13     if s' ∉ stateTable then
14         stateStack.push(s');
15         enabledStack.push(enabled(s'));
16         stateTable.add(s');
```

If multiple transitions are enabled in a state, all possible orderings of executing these transitions need to be considered. This is generally referred to as interleaving, which is the major cause to state explosion in verifying concurrent systems. However, it has been discovered that most of the transition interleavings are redundant with respect to the final verification results, and identifying and removing the redundant interleavings can dramatically reduce the number of states explored. This approach is well known as partial order reduction (POR) [5]. In general, POR works by first computing a sufficient condition for an independence relation of all transitions in a LPN model, and then finding a reduced ample set for each state encountered during the state space search. The independence relation defines which transitions can be executed in different orderings but still lead to the same verification results. This section first reviews the general requirements of independent transitions, it then describes a *behavioral analysis* approach based on analyzing the abstract state space of individual components computed by a compositional reachability analysis method [18]. This approach allows much more refined independence relations to be extracted compared to the static approaches that is based on analyzing the syntactic structures of the high-level descriptions, and therefore can lead to more reduction in state space.

A. General Definition of Independence Relations

The state space of a concurrent system often contains many execution paths that correspond to different execution orderings of transitions in different parts of the system [8]. If transitions are independent, then their executions do not interfere with each other, which means that changing the execution orderings does not modify their the system behavior [3]. In this case, it is sufficient to consider one execution ordering. The notion of independence transitions similar to those in [8], [9], [5] is formalized in the following definition.

Definition 3.1: An independence relation $I \subseteq T \times T$ is a symmetric and antireflexive relation over transitions in T such that for each $(t_1, t_2) \in I$, the following conditions need to hold in every state $s \in S$.

1) If $t_1, t_2 \in enabled(s)$, then $t_2 \in enabled(t_{1(s)})$.
2) If $t_1, t_2 \in enabled(s)$, then $t_1 \in enabled(t_{2(s)})$,
3) If $t_1, t_2 \in enabled(s)$, $t_1(t_2(s)) = t_2(t_1(s))$,

The dependence relation $D = T \times T - I$.

Intuitively, independent transitions cannot disable each other, and the execution of independent transitions in different orders is indistinguishable. Additionally, the independence is defined for transitions enabled in a state. It is possible for two transitions not enabled in every state. In a state where they are not both enabled, they are regarded as independent in that state by the definition. This definition, however, do not affect the verification results as partial order reduction is only applied to independent transitions that are both enabled.

The following section describes how to derive an independence relation from LPN models via a behavioral analysis approach.

B. Behavioral Analysis Approach

The static analysis approach computes independence relations from a LPN model based on conservative but easy-to-check conditions as shown in the previous section. Using these conditions, which are typically carried out statically, often fails to find a lot of the independence between LPN transitions, thus leading to exploring more transition interleavings than necessary [16]. For example, consider two transitions t_1 and t_2 where $Assign(t_1) = \{array[i] := 2\}$ and $Assign(t_2) = \{array[j] := 1\}$. If $i \neq j$ in every state, these two transitions are independent. However, it is difficult to determine whether $i \neq j$ holds in every state just by looking at the LPN syntactic structures as generally $i \neq j$ can only be known after the whole state space is searched. To ensuring the sound verification results, these transitions are always assumed to be dependent if their independence is not known for sure. These potentially false dependencies may lead to states that could be reduced otherwise to be generated. These two simple examples indicate the deficiency of the static analysis approach.

In order to deal with this problem, a behavioral analysis approach is developed which aims to determine the independence between transitions from the over-approximate state space models computed by using a compositional reachability analysis method [18]. This paper assumes that a system model $M = M_1||M_2||...||M_n$ is the parallel composition of components $M_i(1 \leq i \leq n)$. Compositional reachability analysis constructs the state space for each component $M_i(1 \leq i \leq n)$ from an under-approximate context, and gradually expands its state space by including all state and transitions allowed by its neighboring components [18]. During this process, constraints are used to exchange interface information among components so that each component can determine which transition are allowed in a state based on the constraints on its inputs. The initial constraints are empty for the input of each component. With these initial input constraints, some components M_i may be able to fire some transitions on its output, and therefore produce new output constraints. Since the output of M_i may be the inputs of some other components M_j, then the output constraints from M_i are used as the input constraints of M_j. This, in turn, may allow some transitions to become enabled in M_j leading to more states and state transitions found. This process continues expanding the component state spaces by exchanging constraints until the output constraints produced by each component could not change anymore.

For example, Fig 1(a) shows a simple asynchronous circuit which is partitioned into three components M_1, M_2 and M_3. Each component has one initial state s_0 with the initial variable values. For M_1 and M_2, no transitions are enabled in the initial state because none of these transitions satisfies the initial constraint. For M_3, the initial constraint allows transition $t_1 1$ to be enabled. After executing this transition, a new state is reached. Fig 2(a)-(c) shows the partial snapshots for this process and Fig 2(d) shows the final result for compositional reachability analysis. More details about this method can be found in [18].

According to Definition 3.1, dependence relation is the complement of the independence relation. Therefore, after

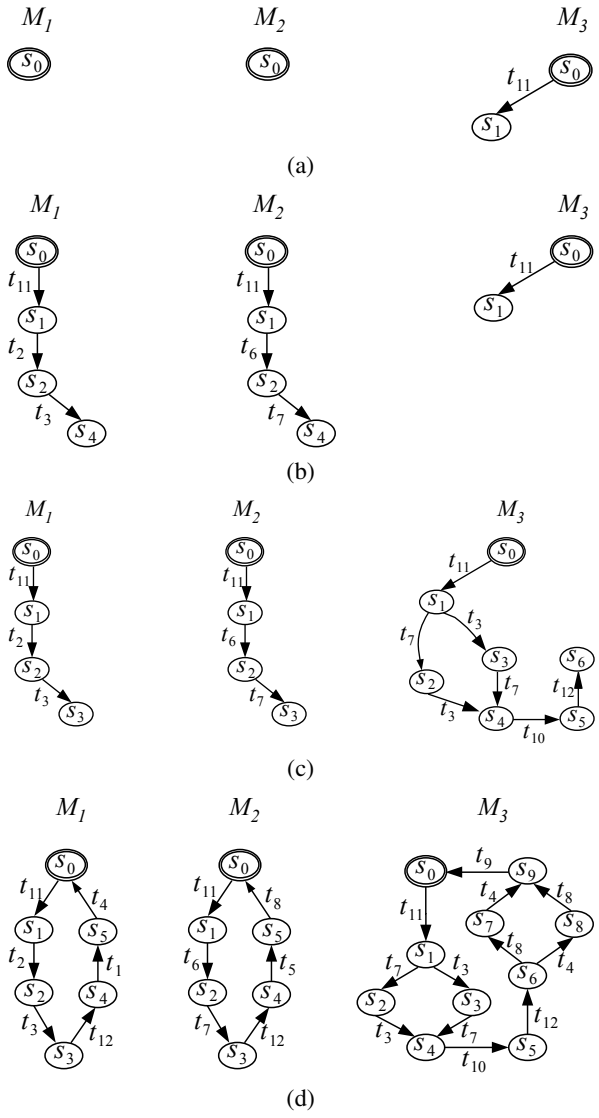


Fig. 2. (a)-(c) Snapshots of the state graphs generated during the compositional reachability analysis. (d) The state graphs generated when the compositional reachability analysis is done.

the component state graphs are generated by the compositional reachability analysis, dependent transitions are found by checking if there is a violation of any condition in Definition 3.1 in any state in any component state graph. More specifically, two transitions t_1 and t_2 are dependent with each other if in any component state graph $G = (init, S, R)$, there exists a state s such that one of the following conditions holds.

1) $\exists(s, t_1, s') : t_2 \in enabled(s) \wedge t_2 \notin enabled(s')$,
2) $\exists(s, t_2, s') : t_1 \in enabled(s) \wedge t_1 \notin enabled(s')$,
3) $t_1, t_2 \in enabled(s) \wedge t_1(t_2(s)) \neq t_2(t_1(s))$,

In the above definition, two transitions depend on each other if firing one can disable the other one as stated in the first two conditions, or firing these two transitions in different orders leads to different states. Fig 3 illustrates dependent transitions found when one of the above conditions holds. Note that in

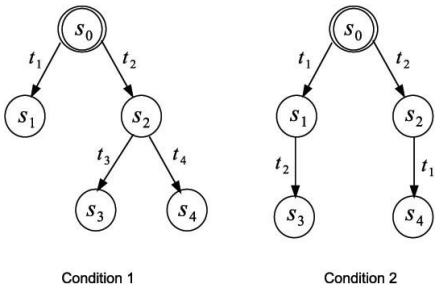


Fig. 3. Examples of dependent transitions that can be found in the state graphs as defined in Definition **??**.

this method if any of these conditions holds for transitions in any state in any component state graph, they are regarded as dependent globally.

In this behavioral analysis approach, much more accurate information on how different orderings of firing transitions affect, for example, whether a transition can possibly be disabled by another transition, is readily available, therefore much more refined dependence relations, and equivalently independence relations, can be derived. This allows many transitions enabled in each state to be removed during state space search, thus leading to enormous reduction in state space, as shown by the experimental results in section V. On other hand, the local state space models generated are over-approximations as these models may contain additional states and state transitions that would not exist if the monolithic state space model is generated for the whole system. This is proved in [18]. Because of these additional states and state transitions, dependence relation found may still be conservative. This is equivalent to saying that some transitions that should be independent may be found as being dependent due to the additional states and state transitions. However, this does not affect the correctness of this method, and it would only cause partial order reduction to be less effective.

IV. Partial Order Reduction

First of all, it is necessary to differentiate the global states of the whole design and the local states of the components of the design. In this method, the LPN of a design is not a single monolithic LPN model. Instead, it is a collection of LPN modules, each of which represents a component in the design. In the rest of this paper, these LPNs are called LPN modules. Each LPN module describes the behavior defined by a design component with respect to the input behavior defined by its neighboring components. Let $M = M_1 \| \ldots \| M_n$ be the LPN of a design where $M_i (1 \leq i \leq n)$ is the LPN for the ith component of the design, and $\|$ is the parallel composition operator for LPN modules [1]. The states of the whole design M are referred to as the *global* states denoted $\vec{s}$, and the states of the individual modules are referred to as *local* states denoted s. The global states for M are not represented as monolithic entities either. Instead, they have similar structure to that of M. More specifically, a global state $\vec{s}$ of M is a n-tuple of the local states $(s_1, \ldots, s_n)$ where each s_i $(1 \leq i \leq n)$ is a local state of LPN module M_i. The global states are found when the state space of the whole design is searched, while the local states are generated during the compositional reachability analysis as described in the last section. For convenience, a predicate $in(\vec{s}, s_i)$ is defined to indicate whether the local state s_i is part of the global state $\vec{s}$. $in(\vec{s}, s_i)$ holds if the local state s_i is part of the global state $\vec{s}$.

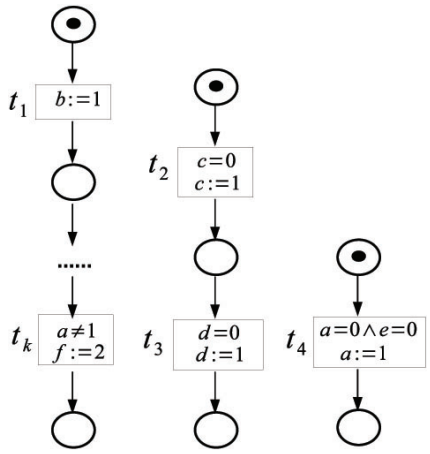


Fig. 4. Examples of dependent and independent transitions' Analysis for Ample Set.

After the independence relation is obtained, the next step is to compute a subset, $ample(\vec{s})$, of the enabled transitions $enabled(\vec{s})$ in each state $\vec{s}$ during the state space exploration. To reduce the complexity of the state space exploration, the size of $ample(\vec{s})$ should be as small as possible, and computing $ample(\vec{s})$ should have low overhead. To preserve the sufficient behavior to verify all properties soundly, four conditions need to be satisfied as described in [5]. Among all the conditions, at-least-one successor and invisibility conditions can be readily satisfied. This section considers dependent-transition and cycle conditions for computing $ample(\vec{s})$.

A. Dependent-Transition Rule

This section presents two conditions to make sure that all essential interleavings between dependent transitions are preserved.

Condition 1 For all enabled transitions $t_1, t_2 \in enabled(\vec{s})$, if $t_1 \in ample(\vec{s})$ and $t_2 \notin ample(\vec{s})$, then $I(t_1, t_2)$ holds.

Intuitively, Condition 1 requires that every transition in $enabled(\vec{s}) \backslash ample(\vec{s})$ is independent on every transition in $ample(\vec{s})$. This condition guarantees that transitions in $enabled(\vec{s}) \backslash ample(\vec{s})$ are still enabled after any transition in $ample(\vec{s})$ is fired. However, this condition alone is not sufficient to guarantee that no states can miss by firing transitions in $ample(\vec{s})$ first. Refer to Fig. 4 for a simple example. In the initial state $\vec{s}_0$, $enabled(\vec{s}_0) = \{t_1, t_2, t_4\}$. Suppose that every transition in $enabled(\vec{s}_0)$ is independent with each other. Therefore, any single transition can be chosen to be included in $ample(\vec{s}_0)$ according to Condition 1, and it can be seen that the other transitions can still remain enabled after firing the transition in $ample(\vec{s}_0)$. First, choose t_1 to

be included in $ample(\vec{s}_0)$. Suppose that another state $\vec{s}$ is reached after firing t_1 and some other transitions where t_k and t_4 are enabled. From Fig. 4 it can be seen that firing t_k and t_4 in different orders may lead to different state spaces since firing t_4 first disables t_k. Now suppose $ample(\vec{s}_0) = \{t_4\}$ and firing t_1 is delayed. This is still valid according to Condition 1. Since transition t_4 fires before t_1, t_k may never get a chance to fire, thus possibly causing certain states not to be found. To avoid this situation, the following condition needs to be satisfied too when computing the ample set.

Condition 2 For each transition $t \in enabled(\vec{s})$, if there is another transition $t' \notin enabled(\vec{s})$ such that $D(t, t')$ holds, then $t \notin ample(\vec{s})$.

Intuitively, this condition requires that firing an enabled transition, if dependent on any other transition that may be enabled in the future, needs to be delayed until its dependent transition also becomes enabled in the same state. This makes sure that the interleavings involving a currently enabled transition and another one to be enabled in the future are not lost. In the example shown in Fig. 4, since $D(t_4, t_k)$ holds for $t_k \notin enabled(\vec{s}_0)$, in the initial state $\vec{s}_0$, $t_4 \in ample(\vec{s}_0)$. In other words, with Condition 2, firing t_4 needs to be delayed until both t_4 and t_k are enabled, therefore firings of t_2 and t_k can be interleaved.

The basic idea behind Condition 2 is that if a currently enabled transition is dependent on another transition that might be enabled in the future, the computed ample set must preserve the interleavings of firing these two transitions whenever possible by putting off firing the currently enabled transition as late as possible.

It is possible that the only ample set that satisfies both conditions is empty. In this case, all the enabled transitions have to be included in the ample set, i.e. if $ample(\vec{s}) = \emptyset$ according to Condition 1 and 2, then $ample(\vec{s}) = enabled(\vec{s})$.

Based on the above discussion, $ample(\vec{s})$ can be computed from $enabled(\vec{s})$ as follows.

1) First, it is partitioned to two subsets $dep(\vec{s})$ and $indep(\vec{s})$. $dep(\vec{s})$ includes all transitions t enabled in $\vec{s}$ such that $D(t, t')$ for some transition t', and $indep(\vec{s}) = enabled(\vec{s}) \backslash dep(\vec{s})$ is the set of transitions enabled in $\vec{s}$ such that each transition $t \in indep(\vec{s})$ does not depend on any other transitions. If $indep(\vec{s}) \neq \emptyset$, we can choose any single transition of $indep(\vec{s})$ to be included in $ample(\vec{s})$.
2) If $indep(\vec{s}) = \emptyset$, one option is to set $dep(\vec{s})$ to be $ample(\vec{s})$.
3) Notice that Condition 2 only requires that a currently enabled transition cannot be included in the ample set if it is dependent on some transition that is not currently enabled. This observation can be utilized to create a smaller ample set from $dep(\vec{s})$ as follows. First, make $ample(\vec{s}) = dep(\vec{s})$. Then, remove every transition t from $ample(\vec{s})$ such that t is dependent on some transitions that is not currently enabled. The resulting $ample(\vec{s})$ still satisfies the above two conditions.
4) If the resulting $ample(\vec{s})$ is empty, finally set $ample(\vec{s})$ to be $dep(\vec{s})$.

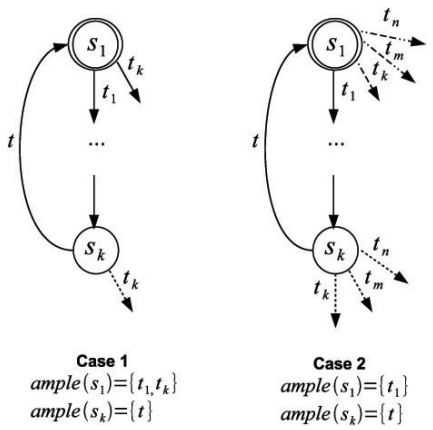


Fig. 5. Examples of two cases for Cycle Analysis.

B. Cycle Rule

Only considering the above conditions is not enough as it may lead to some transitions that are never fired when a cycle is formed. For example, suppose there exist two independent processes A and B where all transitions in A are independent with all transitions in B. Also suppose that transitions in process A are always fired first. If cycles are always formed eventually after firing transitions from process A successively, then transitions from Process B will never get a chance to be fired.

The original cycle property in [5] requires that at least one state in each cycle during state space exploration to be fully expanded so that the reduced transitions are put back to the ample set. However, this requirement is often too conservative as the transitions put back to the ample set may have already been considered in other states in the same cycle, therefore leading to redundant states generated. In our approach, when a cycle is formed, instead of adding all reduced transitions blindly back to the ample set, we check if these transitions are necessary to be added back.

For convenience, let $reduced(\vec{s}) = enabled(\vec{s}) \backslash ample(\vec{s})$ represent the reduced transitions at state $\vec{s}$. Suppose firing an enabled transition $t \in ample(\vec{s}_k)$ will form a cycle to state $\vec{s}_1$ as shown in Fig. 5. The following two cases are considered.

Case 1 If $reduced(\vec{s}_k) \subseteq ample(\vec{s}_1)$, or there exists at least one transition $t' \in ample(\vec{s}_k)$ such that firing t' does not form a cycle, then it is not necessary to add back $reduced(\vec{s}_k)$ to $ample(\vec{s}_k)$.

Intuitively, case 1 deals with the reduced transitions which can be omitted when a cycle forms. In this case, if $reduced(\vec{s}_k) \subseteq ample(\vec{s}_1)$, it indicates that transitions in $reduced(\vec{s}_k)$ are included in the ample set of the next state and can be fired in $\vec{s}_1$. Therefore, it is unnecessary to consider these transitions again in state $\vec{s}_k$. In other words, state $\vec{s}_k$ does not need to be fully expended when the transitions in $reduced(\vec{s}_k)$ have been considered in some state in the same cycle. Fig. 5-case 1 shows a simple example where the transitions in the ample set of each state are represented by the solid arrows and those in the reduced set are represent by

dot arrows. For $\vec{s}_1$, it has been already fully expanded such as $ample(\vec{s}_1) = enabled(\vec{s}_1) = \{t_1, t_k\}$. In the current state $\vec{s}_k$, $enabled(\vec{s}_k) = \{t, t_k\}$, $ample(\vec{s}_k)$ only includes transition t, and firing t forms a cycle. Since the reduced transition t_k enabled in $\vec{s}_k$ is included in $ample(\vec{s}_1)$, it is unnecessary to add t_k back to $ample(\vec{s}_k)$ and consider it again.

A different situation indicated in case 1 is when $ample(\vec{s}_k)$ includes another transition t' whose firing does not form a cycle. Then, we can also omit the reduced transitions in state $\vec{s}_k$ since they are still enabled in the next state after firing t'.

If the conditions defined in case 1 do not exist, then we can check whether adding some of the reduced transitions back are sufficient based on the conditions defined in the following case.

Case 2 If $reduced(\vec{s}_k) \not\subseteq ample(\vec{s}_1)$ and firing any transition $t \in ample(\vec{s}_k)$ forms a cycle, then the following conditions need to be considered:

- If there exists $t \in reduced(\vec{s}_k) \backslash ample(\vec{s}_1)$ such that it is independent with all other transitions in T, then add t only back to $ample(\vec{s}_k)$.
- If no transition in $reduced(\vec{s}_k)$ satisfies the above condition, then a subset $B \subseteq reduced(\vec{s}_k)$ is found first such that for each transition $t \in B$, $D(t, t')$ holds for some $t' \notin enabled(\vec{s}_k)$. Next, for every other transition $t_d \in reduced(\vec{s}_k)$, if $D(t_d, t_B)$ holds for any transition $t_B \in B$, add t_d to B. Finally, if $reduced(\vec{s}_k) \backslash B \neq \emptyset$, add transitions in $reduced(\vec{s}_k) \backslash B$ to $ample(\vec{s}_k)$.
- Otherwise, add all transitions in $reduced(\vec{s}_k)$ back to $ample(\vec{s}_k)$.

The conditions defined in Case 2 aim to finding out what reduced transitions are needed to add back to ample set. Refer to Fig. 5-Case 2 for a simple example. In state $\vec{s}_1$, $enabled(\vec{s}_1) = \{t_1, t_k, t_m, t_n\}$ and $ample(\vec{s}_1) = \{t_1\}$. If the current state during the search is $\vec{s}_k$, $enabled(\vec{s}_k) = \{t, t_k, t_m, t_n\}$, firing transition t from the ample set forms a cycle. Now consider the reduced transition set $reduced(\vec{s}_k)$. Suppose t_k is independent with any other transitions in T. Then, adding t_k only back to $ample(\vec{s}_k)$ would be sufficient since t_m and t_n can still be enabled after firing t_k. If there is no independent transition in the reduced set, the reduced set is partitioned into B and $reduced(\vec{s}_k) \backslash B$ according to condition 2. More specifically, transitions in subset B are dependent either on some other transitions that are not currently enabled, therefore their firings have to be delayed, or on some other transitions in the same set B. In other words, the transitions in $reduced(\vec{s}_k) \backslash B$ are dependent only on each other in the same set, but not any other transitions. Therefore, adding transitions in $reduced(\vec{s}_k) \backslash B$ back to $ample(\vec{s}_k)$ would be sufficient as the transitions in set B are enabled in the next state if firing some transition in $reduced(\vec{s}_k) \backslash B$ does not form a cycle, or transitions in set B are considered again by the cycle rule if firing every transition in $reduced(\vec{s}_k) \backslash B$ forms a cycle.

To illustrate this idea, consider the example shown in Fig. 5 Case 2 where $reduced(\vec{s}_k)$ includes t_k, t_m, and t_n. Suppose $D(t_k, t_m)$ and $D(t_n, t_w)$ hold where t_w is some transition not in $enabled(\vec{s}_k)$, and t_k and t_m are not dependent on any other transitions. Then adding t_k and t_m back to $ample(\vec{s}_k)$ would be enough as firing these two transitions does not disable t_n and there are no other interleavings involving either t_k or t_m that need to be preserved.

Take the circuit shown in Fig 1 as an example. From the state space generated for each component as shown in Fig. 2 by the compositional reachability analysis, there is no dependence among any two transitions. Using the partial order reduction method described in this paper, 12 states are found for this example. We also modeled this example in Promela, and used SPIN to search its state space. In this case, 20 states are found, and there is no reduction. These results are also reported in Table I for the example with the name of fig3a.

V. Experimental Results

The partial order reduction method described in this paper is implemented in an asynchronous system verification tool $Platu$, an explicit model checker implemented in Java. Experiments have been been performed on a number of examples. These examples are asynchronous circuit designs from previously published papers [11], [4], [15], [17], [13]. All these examples are modeled in LPNs and Promela, and experimented with both Platu and SPIN [10]. When running SPIN on these examples, the partial order reduction within SPIN is turned on. This provides a comparison of our approach to the state-of-the-art. All experiments are performed on a Linux workstation with a Intel Pentium Dual-Core CPU and 4 GB memory.

The experimental results are shown in Table I. For each example, its state space is searched by monolithic reachability analysis, SPIN with partial order reduction, and our behavioral analysis approach described in this paper. In the table, the first two columns show the designs and their sizes in terms of number of variables. Since these examples are circuit designs, the variable type is Boolean. For each method used to find the example's state space, the total runtime, memory used, and the number of states found are shown in columns Time, Mem, and S. Runtime is in seconds, and memory is in MB. For all examples, a limit of 5 minutes is imposed. Entries in the table with $-$ indicate the search for that example runs out of time or 2 GB memory space is exhausted.

From table I, it can be seen that SPIN is not able to find any reduction for a number of examples including fig3a, fifo and arb designs with different numbers of components. For other examples, some reduction in state space is found by SPIN, but the reduction is not good enough to allow larger examples to be handled. On the other hand, our approach is far more efficient as it can finish more and much larger examples. However, our approach does not finish for mmu either. This design is highly concurrent and very complex. SPIN is not able to find enough reduction to finish this example, and quickly exhausts 2 GB memory. The behavioral analysis approach, even though not able to finish either, reveals a sufficient independence relation, and finds over 5 million states with about 800 MB memory used. These results show that the behavioral analysis approach indeed is capable of deriving more accurate independence relations from state space models, thus allowing more transitions to be identified as independent.

The overhead of the compositional reachability analysis is relatively small in total runtime. Except for mmu, generally 10

TABLE I

COMPARISON BETWEEN BEHAVIOR ANALYSIS POR METHOD AND SPIN POR METHOD (TIME IS IN SECONDS, AND MEMORY IS IN MBS.). $|S|$ IS THE NUMBERS OF STATES FOUND AT THE END OF REACHABILITY ANALYSIS. ENTRIES FILLED WITH $-$ INDICATES TIME-OUT.

Designs		Method 1:Reachability Analysis			Method 2:POR-SPIN			Method 3:POR-Behavioral Analysis		
Name	$\|V\|$	Time	Mem	$\|S\|$	Time	Mem	$\|S\|$	Time	Mem	$\|S\|$
fig3a	6	0.044	2.7	20	0	2.195	20	0.047	3.292	12
arb1	10	0.056	2.9	52	0	2.195	52	0.043	3.171	31
arbN3	26	0.315	2.4	3756	0.015	2.781	3756	0.109	1.116	356
arbN5	44	8.105	61.538	227472	1.65	71.695	227472	0.281	2.152	5099
arbN7	62	–	–	–	–	–	–	1.825	30.477	90754
arbN9	80	–	–	–	–	–	–	8.376	99.617	398579
arbN11	98	–	–	–	–	–	–	122.347	1360.516	4862988
fifoN3	14	0.119	4.8	644	0	2.195	644	0.047	3.599	51
fifoN5	22	0.733	16.253	20276	0.08	6.593	20276	0.047	3.999	121
fifoN8	34	199.353	845	3572036	30.2	1087.211	3572036	0.062	1.368	286
fifoN10	42	–	–	–	–	–	–	0.078	2.477	436
fifoN20	82	–	–	–	–	–	–	0.253	6.579	1666
fifoN50	202	–	–	–	–	–	–	0.086	14.279	10156
fifoN100	402	–	–	–	–	–	–	5.086	81.922	40306
fifoN200	802	–	–	–	–	–	–	32.665	328.889	160606
fifoN300	1202	–	–	–	–	–	–	137.048	978.714	360906
dmeN3	33	3.589	26.1	$267,999$	0.265	19.706	117270	0.202	1.745	912
dmeN4	44	1235	1032	$15.7M$	15.5	553.421	4678742	0.25	3.685	4495
dmeN5	55	–	–	–	–	–	–	0.437	5.252	15452
dmeN8	88	–	–	–	–	–	–	13.118	174.070	687475
dmeN9	99	–	–	–	–	–	–	49.858	532.353	2471839
tagunit	48	–	–	–	4.37	144.984	786672	0.187	4.897	1103
pipectrl	50	–	–	–	–	–	–	0.468	5.670	4610
mmu	55	–	–	–	–	–	–	–	–	$> 5M$

percent of the total runtime is spent on finding the state space for components, and deriving the dependence information from these state space models. For mmu, the overhead of the behavioral analysis is about 35 percent of the total runtime. This is due to that many components in this design have quite complex interfaces, and a large number of extra states and state transitions are generated for each component during the compositional readability analysis.

VI. CONCLUSION

This paper introduces a new approach to computing independence relations for partial order reduction. Since the analysis is applied on the state space models, the derived independence relations are more refined and accurate, which lead to more effective partial order reduction as shown by the experimental results. In the future, we plan to integrate this approach with compositional verification in order to scale model checking for even larger designs, and also extend the idea presented in this paper to real-time system verification.

REFERENCES

[1] J. Ahrens. A compositional approach to asynchronous design verification with automated state space reduction. Master's thesis, Univ. of South Florida, 2007.

[2] C.Baier and J.-P.Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] C.Flanagan and P.Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages*, pages 110–121, 2005.

[4] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.

[5] E.M.Clarke, O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[6] E.M.Clarke, O.Grumberg, and M.Minea. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[7] G.Gueta, C.Flanagan, and E.Yahav. Cartesian partial-order reduction. In *SPIN Workshop on Model Checking Software*, volume 4595 of *LNCS*. Springer, 2007.

[8] G.J.Holzmann and D.Peled. An improvement in formal verification. In *roc. Seventh FORTE Conf. Formal Description Techniques*, 1994.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, University of Liege, 1995.

[10] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[11] A. J. Martin. Self-timed fifo: An exercise in compiling programs into vlsi circuits. Technical Report 1986.5211-tr-86, California Institute of Technology, 1986.

[12] T. Murata. Petri nets: Properties, analysis, and applications. In *Proceedings of the IEEE 77(4)*, pages 541–580, 1989.

[13] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.

[14] R.A.Thacker, K.R.Jones, C.J.Myers, and H. Zheng. Automatic abstraction for verification of cyber-physical systems. In *1st International Conference on Cyber-Physical Systems*, 2010.

[15] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, 1999.

[16] V.Kahlon, C.Wang, and A.Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proc. Int. Conf. on Computer Aided Verification*, volume 5643 of *LNCS*. Springer, 2009.

[17] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proc. of Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 152–163, 1996.

[18] H. Zheng. Compositional reachability analysis for efficient modular verification of asynchronous designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):329–340, 2010.